NPS CS-92-021

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A VERSION AND CONFIGURATION MODEL FOR SOFTWARE EVOLUTION

Salah M. Badr

Luqi

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

REAR ADMIRAL R. W. WEST JR.
Superintendent

HARRISON SHULL
Provost

This report was prepared for and funded by the Naval Postgraduate School.

This report was prepared by:

Luqi
Associate Professor of
Computer Science

Reviewed by:

C. THOMAS WU
Associate Chairman for
Technical Research

Released by:

PAUL MARTO
Dean of Research

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPS CS-92-021	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Postgraduate School; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A VERSION AND CONFIGURATION MODEL FOR SOFTWARE EVOLUTION

12. PERSONAL AUTHOR(S)
SALAH M. BADR, LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM 05/89 TO 03/90	December 1992	21

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	SOFTWARE EVOLUTION, VERSION CONTROL, CONFIGURATION MANAGMENT, JOB ASSIGNMENT, DESIGN DATABASE,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report introduces both a design database model and a version and configuration model for keeping track of all these overwhelming numbers of software system versions and keep track of which object version belongs to which system configuration in an automated manner transparent to the user. This gives the software development team the chance to concentrate on what is needed to be done to fix or improve system components instead of worrying about managing this enormous amount of data. This paper also shade some lights on our design management and job assignment system (DMJAS) that uses these two models to manage both design data and development team [12].

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: LUQI	22b. TELEPHONE (Include Area Code): (408) 656-2903	22c. OFFICE SYMBOL: CS/Lq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

A Version and Configuration Model for Software Evolution[1]

Salah Badr
Luqi
Naval Postgraduate School
Department of Computer Science
Monterey, California 93943 USA

ABSTRACT

With the complexity of software systems growing every day, more sophisticated development and maintenance environments are necessary to cope with the evolutionary nature of software systems. These systems experience iterative behavior through an endless number of versions to cope with the customer's changing and growing needs and the changing and growing software and hardware technology.

This report introduces both a design database model and a version and configuration control model for keeping track of these overwhelming numbers of software system versions and keep track of which object version belongs to which system configuration in an automated manner which is transparent to the user. This allows the software development team to concentrate on what is needed to be done to fix or improve system components instead of worrying about managing this enormous amount of data. This paper also sheds some lights on our design management and job assignment system (DMJAS) that uses these two models to manage both design data and the development team [12].

KEYWORDS: SOFTWARE EVOLUTION, VERSION CONTROL, CONFIGURATION MANAGMENT, JOB ASSIGNMENT, DESIGN DATABASE, SOFTWARE ENGINEERING.

1. Introduction

In an evolving software system, the design database plays an active role in the development/evolution process. It represents the kitchen in which all kinds of project data are cooked and kept in different forms. The project releases are frozen in a write-protect mode, copies of the project data under development/evolution are kept in a mutable form accessible only to the evolution

1. This research was supported in part by the Army Research Office under grant number ARO-145-91, and in part by the National Science Foundation under grant number CCR-9058453

team, and history files are kept in an archival form (new versions can be added but existing versions cannot be modified after they are committed). This kitchen has to be run under a strict cook that takes care of the coordination and management of the different activities.

Our models of version and configuration control, and design database together with our Design Management and Job Assignment System DMJAS [12] have the following goals [7].

- Minimize the communication time between designers/programmers by keeping the development documents on-line.
- Support planning and scheduling of proposed changes/maintenance by keeping the relationship between different data objects, and managing changes in progress.
- Propagate change consequences to maintain the global consistency of the database.
- Produce nondisruptive status reports for an ongoing project.
- Track the development history by maintaining an up-to-date record of all design decisions and relevant development information.
- Support software reusability to save both cost and effort.

The version control and configuration management mechanism (VCCM) as part of the Design Management and Job Assignment System DMJAS [12] automatically takes care of object versioning and system configurations (releases).

Our model of the design database has the features needed to support all of the above requirements except the last one. A separate database called the software base contains all reusable software components and the retrieval system that finds existing components that match given specifications [21].

An evolving system goes through successive iterations during its development to meet the customer's real needs. It also continues to evolve during its life time to cope with customer's changing requirements and the advances in both software and hardware systems besides the normal maintenance activities for bug fixing and upgrading. This process of evolution is inherent to almost each software system. If S is the intended final version of the software system, then each successive iteration of this system can be viewed as an element of a sequence Y_i where: $\lim_{i \to \infty} Y_i = S$. Each system iteration (version) Y_i is modelled as a graph $G_i = (V_i, E_i)$, where: V_i is a set of vertices. Each vertex represents an atomic object or a composite object modelled as

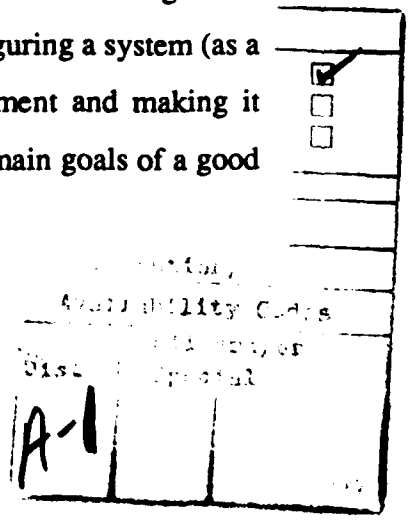
another graph, and E_i is a set of edges. Each edge represents one of the relations "part_of" or "uses" between two different objects of those represented by the set of vertices V_i. This mathematical formulation is the basis of our design database data model and also the basis of our computed labelling function described in sections 3 and 4 of this paper.

2. Previous Work

As indicated in [13], version control and configuration management is one of the fields in software engineering that has received much discussion and many proposals for proper version and configuration models in different domains, but little has been implemented, and much remains to be done in developing techniques for ensuring the consistency of configurations and space efficient algorithms for version management.

The versioning process as described in [15] is divided into two main models: the conventional version oriented model VOM, and the change oriented model. Our VCCM mechanism supports change oriented versioning, where a new system configuration is generated each time we complete a set of changes (an evolution step) and orthogonal to that a new version of each object involved in the change is also generated. The three main aspects of organizing software objects defined in [16], evolution, membership and composition, are similar to the underlying concepts used by our mechanism; the difference is that we use composite objects to represent the membership organization and a generalized form of the composite object to define the composition organization. This same structure represents configurations of systems and their subsystems.

Our concept of composite entities and its generalization to fit system configurations is also similar to that used in PACT [14]. Our system uses a computed labeling function and a single versioning mechanism for automatically versioning individual objects and configuring a system (as a composite object). Simplifying version control and configuration management and making it transparent to the user without requiring his/her intervention are two of the main goals of a good version control and configuration management system as set forth in [3].



Our system takes care of planning, scheduling, status accounting and auditing of the changes via explicit representations of steps as well as versions. Each step has a unique step number and all the relevant information such as dependent modules, affected modules, who made the changes and when, and the current status of a step in addition to a description of the motivation for these changes. This enables the system to answer questions similar to those mentioned in [17] such as: what changes were made in step #X, what components were affected by this change, what changes were made to the system after a certain date, and so on.

3. Design Database Model

The data in the design database is modeled as a graph in which the nodes represent uniquely identified versions of objects (atomic or composite) and the arcs represent relationships between these objects. The graph also represents the organization of the software products into separate concerns. Software can be decomposed into vertical and horizontal structures [11].

The vertical structure forms a hierarchical structure with the design database as the root of the hierarchy. The first level represents the different systems/prototypes included in the design database. The second level represents the different variations of each system/prototype. Variations represents alternative choices, which may correspond to different formulations of the requirements in the context of prototyping, or different kinds of underlying system software (operating system, window manager, etc.) in the context of product releases. The third level represents the different system versions (configurations) for each variation. A linear sequence of versions represents the evolution history of a particular variation. The fourth level represents the different modules comprising each version (source code module, specification module, requirement module, etc.). From the fifth level on is the decomposition of each composite module to its sub-modules, down to the leaf nodes which must be atomic. Figure 1 shows a graphic representation of the design database hierarchy. The objects in the vertical hierarchy are linked together through the inverse relations "*part-of*" and "*parent*" between a composite object and its components.

The horizontal structure has two different views: the first represents dependencies between entities that are part of the same release. The main relations in the first view are "*uses*" and "*used*

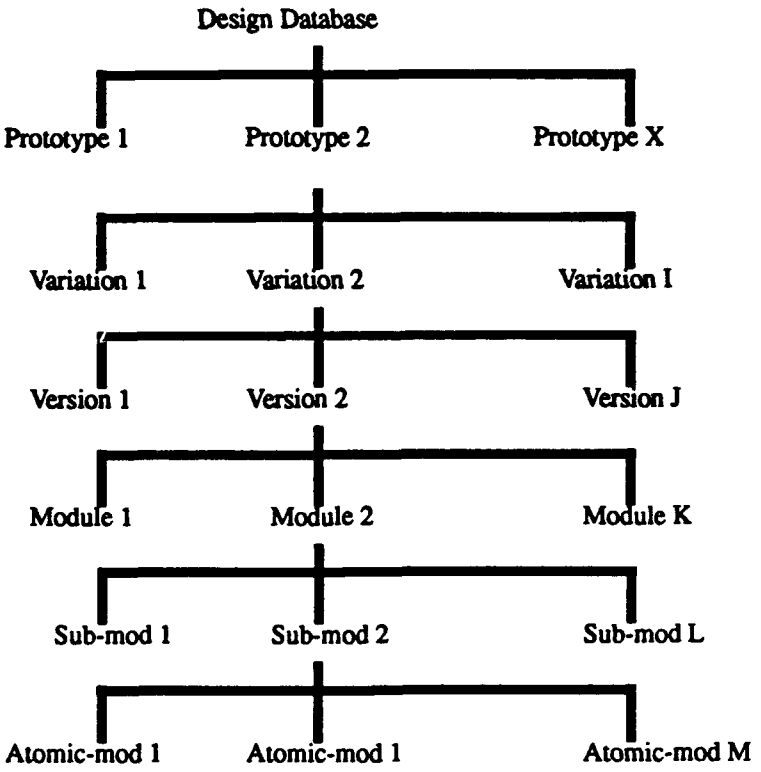


Figure 1 The Design Database Hierarchy Structure

by". The second view represents derivation relationships between different versions of the same object via the relations "*previous*" and "*next*" to facilitate navigation through an object history.

The design database is divided into two main parts:

- *Shared data space* where frozen versions of software objects are stored, and
- *Private workspaces* where new software are developed.

3.1 Shared Data Space

The shared data space is the repository that keeps all of the verified software objects (versions or configurations) [5]. The versions in the shared data space are frozen and may not be changed under any circumstances. Any changes to any of the objects must be done in the context of an evolution step, authorized by the management, and completion of such a step can only add

new versions to the shared data space. The shared data space contains the public releases of the software objects. Mutable copies of these objects can only be obtained as part of an evolution step controlled by the Design Management and Job Assignment system. The relations between the objects in the database are kept as attributes of each object.

3.2 Private Workspaces

Since the data in the shared data space is frozen and may not be changed, the private workspaces are used for production of new versions of existing objects or adding new objects to existing software systems which in turn produce new versions of the software system.

The private workspaces contains copies of the specific versions of the software to which the changes are to be implemented (base versions). Only the designer responsible for an evolution step has access to a mutable copy of the base version in that designer's private workspace, and the designer can modify those objects only via the tools provided by the Computer Aided Prototyping System (CAPS). The process of copying objects to and from the designer's workspace is done automatically by the Design Management and Job Assignment System DMJAS [12], and these objects continue to be under its control until either all the changes are done and the DMJAS commits them to generate the new version (when a mutable version in the design database is committed by the DMJAS, it becomes an immutable version in the shared data space), or if the changes are suspended/abandoned then current copies of the mutable versions are appended to the log associated with the step for future references.

4. Version and Configuration Management Mechanism

In the context of an evolving system, an object is a software component that is subject to change; objects can be either composite or atomic. Objects can be changed only by creating new versions. Change requests submitted by customers or suggested by system engineers trigger *evolution steps* [8]. Applying an evolution step to a specific version of a software object produces a new version of that software object. An evolution step may be either atomic or composite. An atomic evolution step produces at most one new version of an atomic system component, while a

composite evolution step produces a new version of a composite component and its substeps produce new versions of the subcomponents of the composite object [8]. In either case (atomic or composite), every evolution step is part of a transaction that produces a new version of the entire software system under development. A transaction consists of an evolution step that updates an entire prototype, together with all levels of associated substeps. This structure provides accountability and implies that the version number of a software system is greater or equal to the version number of any of its components.

Software evolution steps are created when changes are proposed, and become part of the online representation of the work plan when they are approved. Steps become part of the work schedule when they are bound to specific versions of their input objects which triggers their automatic assignment by the DMJAS to specific designers. Figure 2 depicts the graph representation of the relation between system versions and the corresponding evolution steps. Step S_k is applied to version $V_{i,j}$ of a software object (where "k" is the step number, "i" represents the variation number and "j" represents the version number along one variation) producing version $V_{i,j+1}$. Variations are represented as partial paths in the graph, applying step S_{k+1} to $V_{i,j+1}$ produces the version $V_{i,j+2}$ on the same variation line. Applying step S_{k+2} to $V_{i,j+1}$ produces a new variation i+1 with version $V_{i+1,j+2}$. Applying step S_{k+3} to $V_{i,j}$ produces another new variation i+2 with the version $V_{i+2,j+1}$.

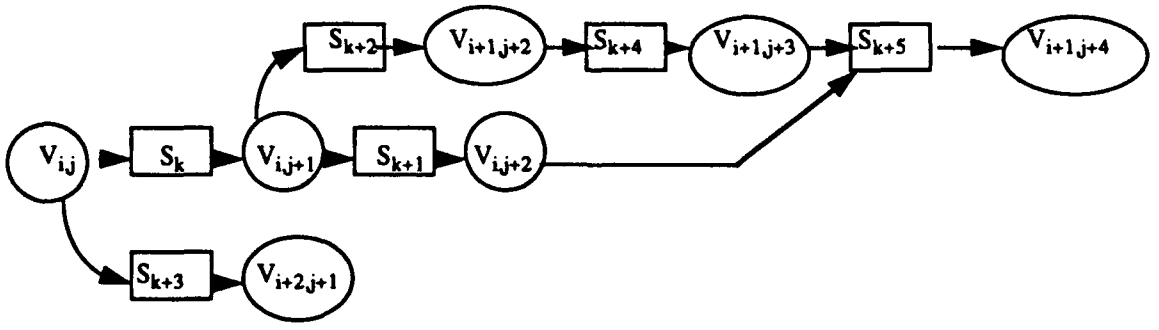


Figure 2 The relation between system versions and evolution steps

The graph can also include dependencies between the modified versions and versions of other objects that are not modified by the step, such as specifications of other modules. For simplicity links of this type are not shown in figure 2.

Notice that, in case of a split that creates a new variation, it is the order of the steps rather than the version number of the base version that decides the variation number (i. e., Step k+2 created the new variation i+1 and Step k+3 created the new variation i+2 despite the fact that the first is applied to version j+1 and the second is applied to version j). Thus the variation numbers capture the chronological order in which the variations were created. Step numbers are assigned in increasing order when the steps are created. Steps can be carried out concurrently and asynchronously, and the order in which they actually start or complete their implementation phases can differ from the order in which the steps are added to the schedule.

4.1 Version and Variation Numbering

As soon as the step number is assigned and the input base version is bound, the system assigns the version and variation number of the output object for the step. The variations are assigned successive numbers beginning with 1 for the initial variation. Versions along each variation are assigned successive numbers starting with 1 at the root version of the initial variation. This means that the new version number is the base version number plus one, while the variation number has two possibilities: the first possibility is to keep the base version's variation number at the time the step is scheduled. This occurs when the base version is the most recent version on its variation line at the time the step is scheduled. The other possibility is to use the "next" variation number, which is the highest variation number plus one. This labeling function is the same for both atomic or composite objects (the entire software system is represented as a composite object). We formalize this labeling function as follows:

a. The ordering of version identifiers on the same variation line must be the same as the serialization order of the changes that produced the corresponding versions.

predecessor (V-new) := exists (V-base) & $\neg$ exists (successor (V-base)) $\rightarrow$

version (V-new) := version (V-base)+ 1, variation (V-new) := variation (V-base)

For the initial version of a newly created prototype:

$\neg$ exists (predecessor (V-new)) $\rightarrow$ version (V-new) := 1, variation (V-new) := 1

b. Changes to versions that are not the most recent on their variation line split off a new variation line and produce an alternate version on the new variation.

predecessor (V-new)= exists (V-base) & exists (successor (V-base)) $\rightarrow$

version (V-new) := version (V-base)+ 1,

variation (V-new) := Next-variation,

Next-variation := Next-variation + 1

where Next-variation is a global variable for each object graph that holds the number of the next variation to be created in this graph, and initially has the value 2.

- In the case of a merge of two variations:

a. If both the base versions are the most recent on their variation lines, then the new version number is the greater version number of the base versions plus one. The new variation number is the variation number of the one that has the greater base version number of the merging variations (to keep it a unique number).

{V-base1, V-base2} $\subseteq$ predecessor (V-new) & $\neg$ exists (successor (V-base1)) &

$\neg$ exists (successor (V-base2)) $\rightarrow$

version (V-new) := max (version (V-base1), version (V-base2))+ 1,

variation (V-new) := variation (max (version (V-base1), version (V-base2)))

b. If only one of the merging versions is the most recent on its variation line, then the result of the merge version is the next version on this same line.

{V-base1, V-base2} $\subseteq$ predecessors (V-new) & $\neg$ exists (successor (V-base1)) &

exists (successor (V-base2)) $\rightarrow$

version (V-new) := version (V-base1)+ 1,

variation (V-new) := variation (V-base1)

c. If both the base versions are not the most recent versions on their variation lines, then the new variation number is the "next" variation number and the version number is the version number of the version of the first base version plus one.

{V-base1, V-base2} $\subseteq$ predecessor (V-new) & exists (successor (V-base1)) &

exists (successor (V-base2)) $\rightarrow$ version (V-new) := version (V-base1) + 1,

variation (V-new) :=Next-variation,

Next-variation := Next-variation + 1

This labeling function allows a version to belong to more than one variation which is a necessary modification to [8] to simplify the process of tracing the development history of a version and to keep a logical and realistic development history.

4.2 Configuration Control Model

As mentioned in section 3 above, the configurations of the different software systems/prototypes are represented by a hierarchical structure according to the levels of the decomposition of each system. This hierarchical structure is a directed acyclic graph in our case with its nodes representing the different components of the system and the edges representing the relations among these components which are "part-of" and "uses". An example of a system is given in section 5 to clarify this configuration graph. An evolution step producing new versions of some of the system components eventually leads to producing a new version of the whole system. This is done by propagating the version numbering recursively from the leaf nodes to their parents and all the way up to the root of the tree which represents the system under evolution. This is formalized as follows:

For i= N to 2

exists (new-version (component (i))) & component (i) part_of component (i-1) &

$\neg$exists (new-version (component (i-1)) $\rightarrow$ new-version (component (i-1))

Where N is number of levels in the configuration graph in which the root is level number 1.

4.3 Committing Evolution Steps

An evolution step can only be committed if all its sub-steps and induced steps are committed [8]. This means that the transition to the "commit" state of a top level evolution step is triggered by the commitment of the last substep in its decomposition. Since the Design Management and Job Assignment System is automatically generating both the primary and induced substeps of a composite step, assigning them, adding and deleting input objects for steps in progress, processing the commit command for each substep, then it has the complete knowledge required to trigger the commitment of the top level evolution step. Committing the individual steps is done by copying their output components to the shared data space of the frozen versions and making these versions visible only to those designers who are assigned steps that use those objects as secondary input objects, while committing the top level evolution step is done by making these new frozen versions visible for public use.

An overview of how the steps are assigned to the members of the design team may clarify the order of committing those steps. The Design Management and Job Assignment System analyzes the primary inputs of the top level evolution steps, finds the induced changes needed to propagate the required changes of the primary inputs, then it creates a hierarchy of steps according to the relations "part_of" and "uses" between the primary and induced inputs. A step can only be assigned if all those steps it depends on are committed, which guarantees the consistency of the software system via propagating the changes and minimizing rollbacks.

4.3.1 Committing Atomic Steps

As defined above an atomic evolution step produces at most one new version of an atomic system component. The output of an atomic step is either a new version of its input object or an initial version of a new object to be added to the system. In either case this object is configured using the labeling function defined in section 4.1, and copied to the shared data space as frozen version. This new frozen version is made visible only to the members of the design team who may use it as a secondary input to their assigned steps. This new version is also mapped to the new configuration graph replacing its previous version or creating a new node in the graph if it is an

initial version of a newly added object. Committing an atomic step also releases all the dependency links between this step and those steps that use this step.

```
commit_step(s: step_id)
WHEN s IN current_step.id
    Commit (s) = Configure (s.output), copy(s.output, shared_data_space)
    OTHERWISE REPLY EXCEPTION no_such_substep
```

4.3.2 Committing Composite Steps

Having the hierarchical structure of both the developed system and the top level evolution steps (the transaction that consists of an evolution step that updates an entire prototype/system) in mind facilitates the understanding of how the commitment of a composite evolution step takes place. First the output of the composite step is copied back to the shared data space, configured, then the "part_of" links between this component and its parts are updated especially for those components that have new versions created as an output of the substeps of this composite step. Going all the way up the hierarchy, the last composite step is the top level step which indicates that all the steps have been committed and that it is time to release the resulting system for public use. For this reason the top level step has something more than each composite step. It completes the configuration graph by updating the "previous" and "next" links for each object in its horizontal graph. This means that the newly created version is connected to its base version for history tracing. Finally, it makes the new version of the software system visible to the user world as a new system release.

5. A Version Control and Configuration Management Example

Assume we have a system Sys as in figure 3 below that consists of three main modules Ma, Mb and Mc. Ma consists of two objects Oa, Ob and Mb is atomic while Mc consists of Oc, Od, and Oe. As shown in figure 3 below, each component has a specification module and an implementation module. The composite components have two extra modules: the graph module and the postscript module (which is not shown in the graph for simplicity). The relations "used-by" and "part-of" in figure 3 are defined as follows: a thin arrow from A to B means that A used_by B,

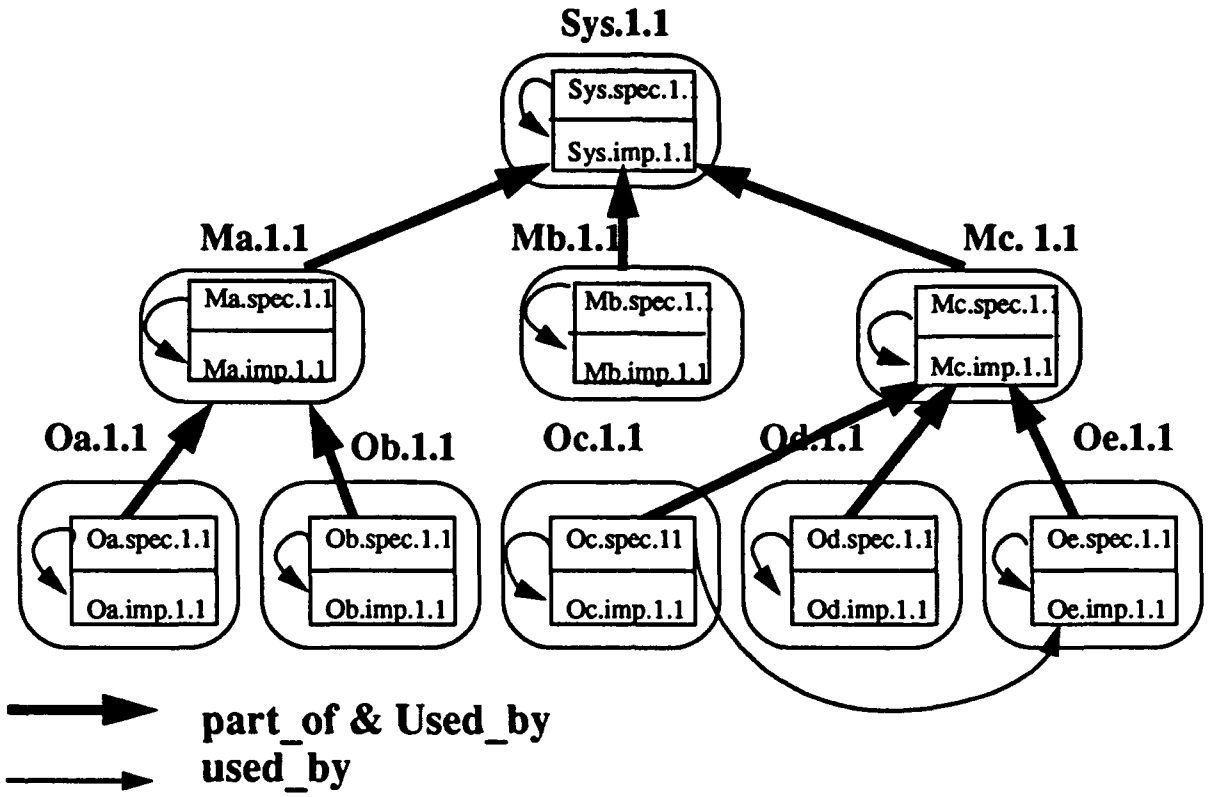


Figure 3 A given system version 1.

while a thick arrow from A to B means A part_of B & A used_by B. This means Ma part-of Sys, Mb part-of Sys, Mc part_of Sys, Oa part-of Ma, Ob part-of Ma, Oc part-of Mc, Od part-of Mc, Oe part-of Mc. The "part-of" relation also implies a "used_by" relationship between the specification modules of the children components and the implementation module of their parent components. The used_by relation is defined as follows: Oc used_by Oe, and every specification module of a component is used_by its implementation module such as S.spec used_by S.imp, Ma.spec used_by Ma.imp, etc.

1. Let us assume that three evolution steps are created by the designers. The first Step is S1 with primary input Ma.spec, no secondary inputs, base version is Sys.1.1. The second step is S2 with primary input Mb.imp, secondary input Mb.spec, and base version is Sys.1.1. The third step is S3 with primary input Oc.spec, no secondary input, base version is Sys. The system will assign a unique number to each of the created steps and the steps status are initialized to "proposed".

2. The manager reviews the proposed steps, adds any management constraints such as priority, precedence, estimated time for completing the step, and deadline. The manager can also modify any of the inputs to the step through the commands add_input or delete_input. When the manager approves the step, its status is changed from proposed to "approved".

3. Changing the step status to "approved" triggers the analysis process that analyzes the relations between the primary inputs and the rest of the system components, determines the induced changes that have to be made to propagate the required changes, then builds the dependency graph between both the primary and induced inputs (those modules that use the output of the step which can be approximated by those modules that uses the input of the step). The analysis process is done as follows:

a. The system will examine the primary input modules with respect to "used_by" relationship to find out what other modules have to be changed to reflect and propagate the required changes (induced changes). By examining the primary input to step 1 the system should find out that the only modules that use Ma.spec are {Ma.imp, Sys.imp}. The same analysis is done for steps two and three. The resulting two induced sets are {} and {Mc.imp, Oc.imp, Oe.imp}.

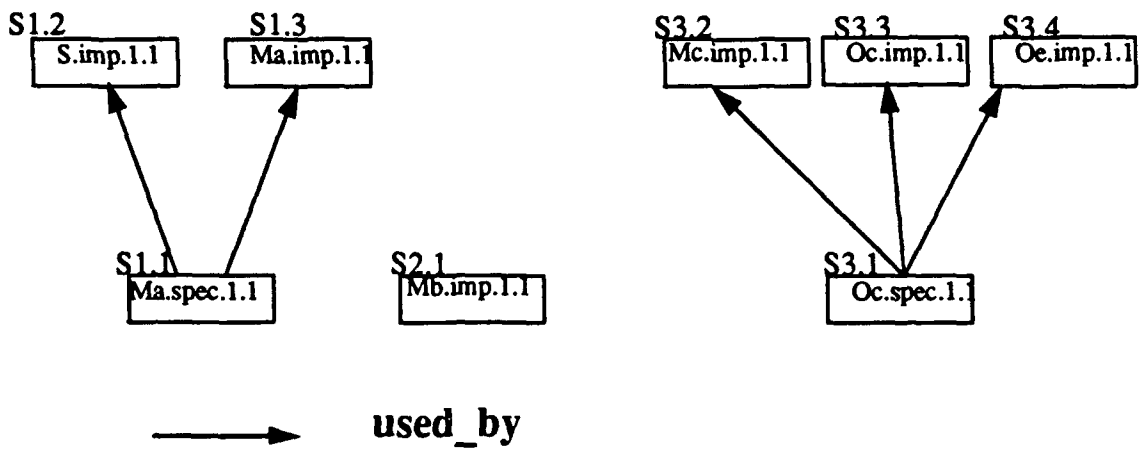


Figure 4 The dependency graph

b. The dependency graph between each step's primary input and its induced inputs is built as shown in figure 4, and a copy is sent to the manager for validation. This dependency graph is an

acyclic graph used for scheduling estimation and also used by the job assignment mechanism for concurrent assignment.

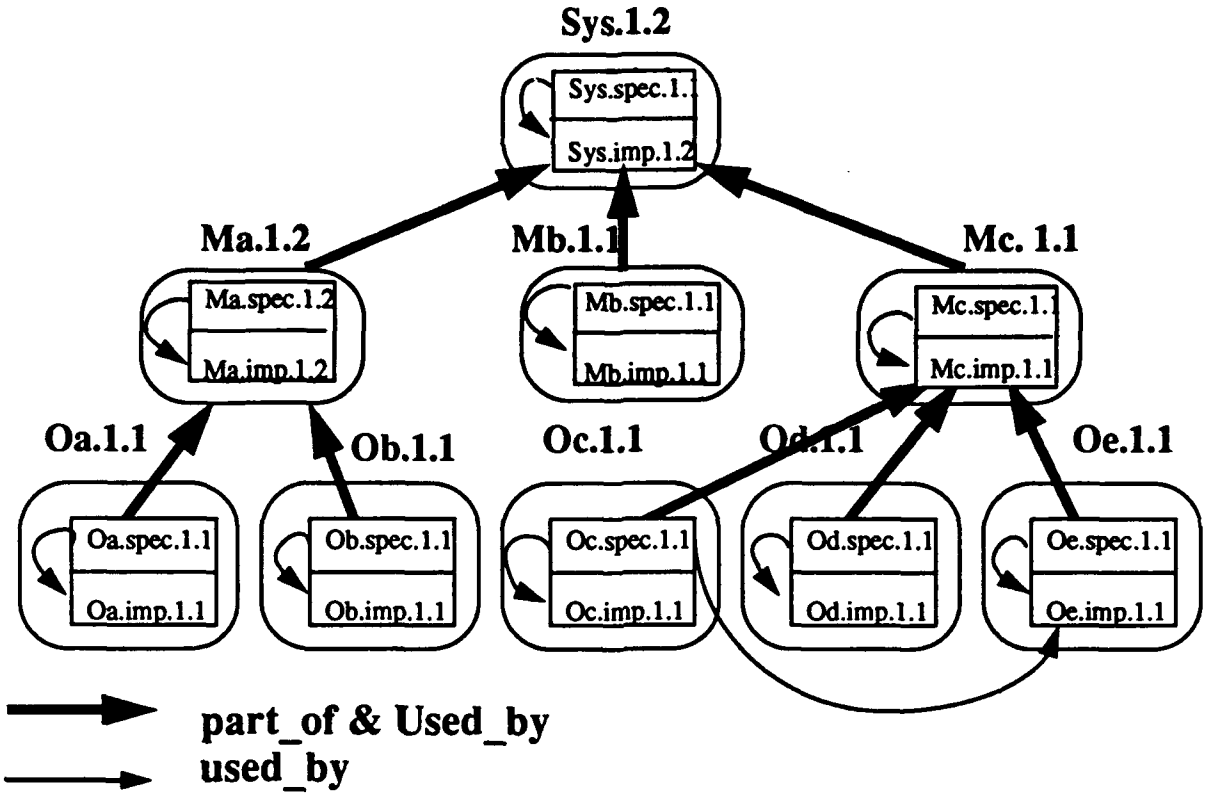


Figure 5 Version 2 of the given system.

4. After reviewing the dependency graph the manager either decides to edit it [add/delete edges or nodes] or to schedule the steps. When the manager decides to schedule the steps, the scheduling and job assignment mechanisms is triggered. The scheduling mechanism works as follows:

a. A substep of the parent top level step is created for each component in the dependency graph. This means steps S1.1, S1.2, and S1.3 are created with the primary inputs Ma.spec.1.1, Sys.imp.1.1, and Ma.imp.1.1 respectively; step S2.1 is created with the primary inputs Mb.imp.1.1; steps S3.1, S3.2, S3.3, and S3.4 with primary inputs Oc.spec.1.1, Mc.imp.1.1, Oc.imp.1.1 and Oe.imp.1.1 respectively. Induced steps inherit the version bindings of their inducing steps as well as their precedence, priority, and deadline if any exist.

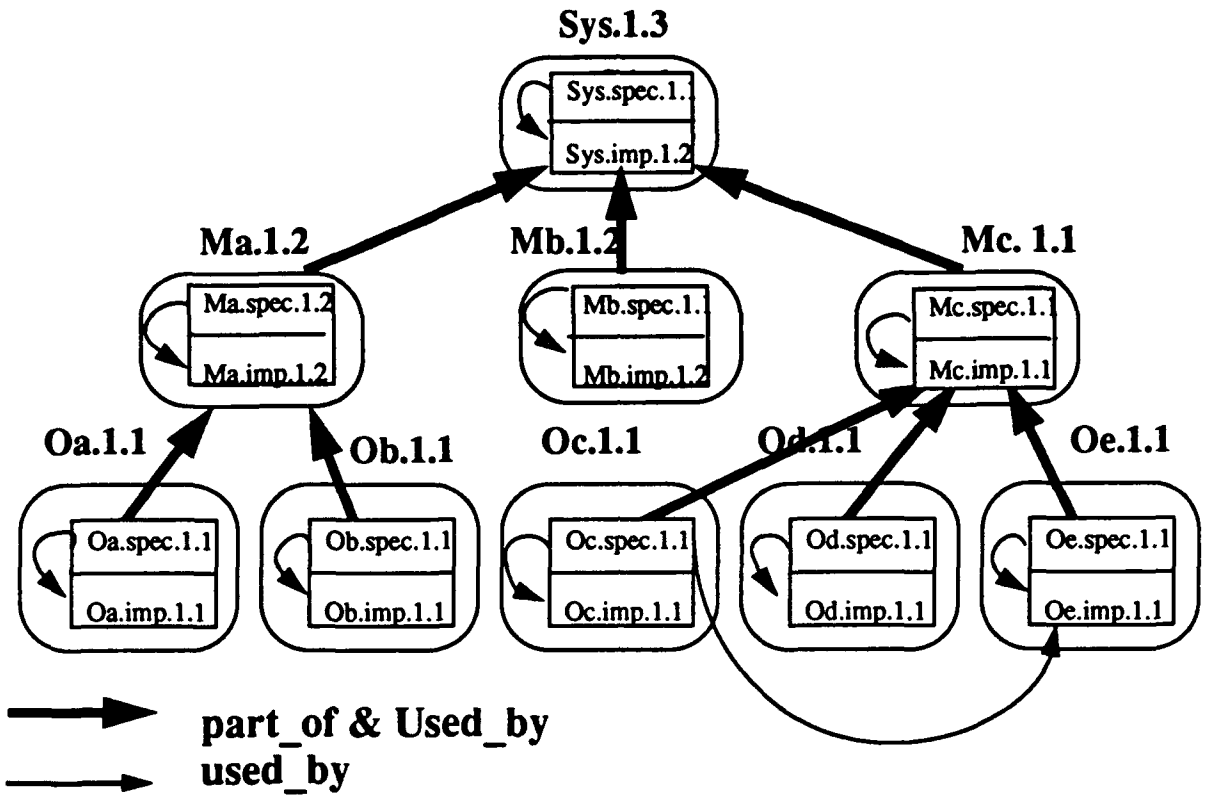


Figure 6 Version 3 of the given system.

b. The DMJAS performs the serialization and assignment of the different steps to the available designers. The serialization and assignment policies as well as the scheduling and assignment mechanisms are explained in [12]. Let us assume that the serialization of the three steps is S1, S2, and S3. This means that the commitment order of the three steps is in the same order, i.e., S1 produces version 2 of the system on variation 1, S2 produces version 3 of the system, and S3 produces version 4 of the system on the same variation

5. Now committing S1 means that its three substeps are committed, S1.1 producing Ma.spec.1.2, S1.2 producing Sys.imp.1.2, and S1.3 producing Ma.imp.1.2. This produces the new version Ma.1.2 of the composite component Ma. The new version of Ma and the new version of Sys.imp are also automatically propagated to produce the new version Sys.1.2 of the whole system as shown in figure 5. This changes the base version binding for step 2 to be Sys.1.2.

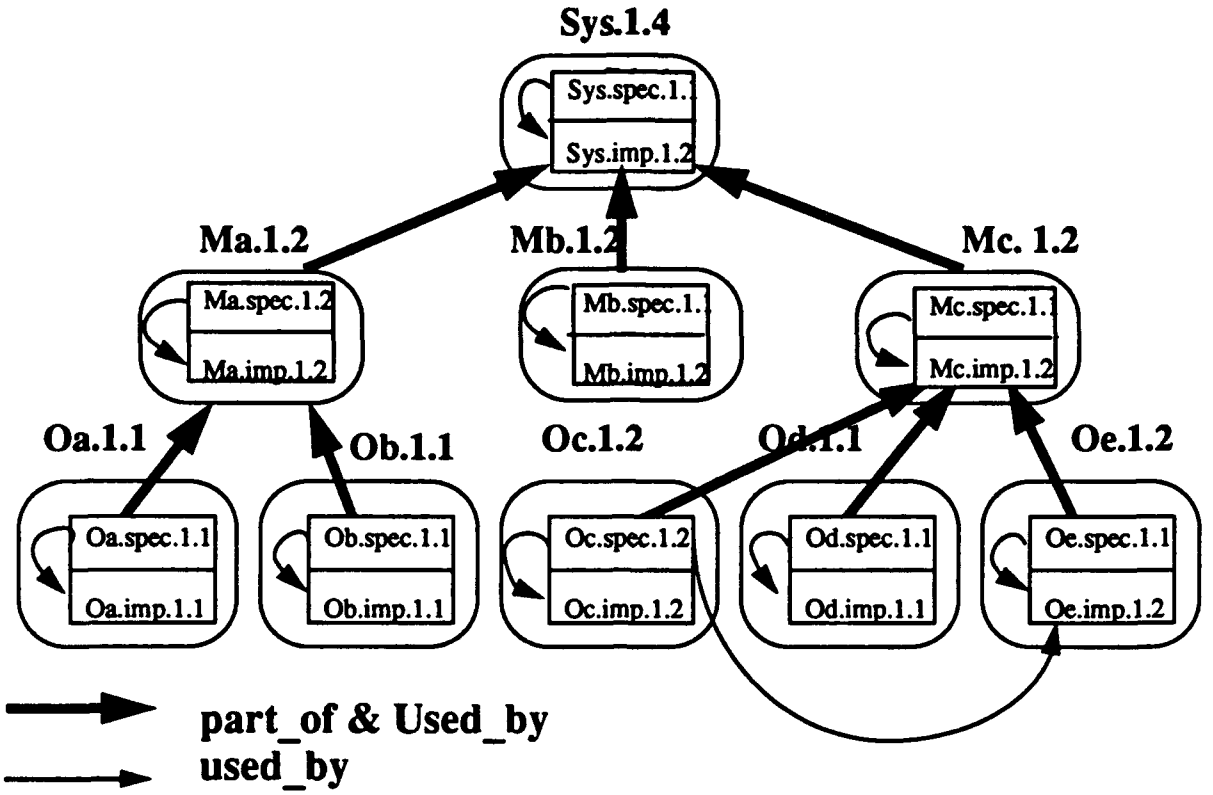


Figure 7 Version 4 of the given system.

6. Committing S2 means that its substep S2.1 is committed producing Mb.imp.1.2. This produces the new version Mb.1.2 of the component Mb. This is automatically propagated to produce the new version Sys.1.3 of the whole system as indicated in figure 6, and change the base version binding for step 2 to be Sys.1.3.

7. Committing S3 means that its four substeps are committed, S3.1 producing Oc.spec.1.2, S3.3 producing Oc.imp.1.2, which automatically produces the new version Oc1.2 of the object Oc; S3.4 producing Oe.imp.1.2 which automatically produces the new version Oe1.2 of the object Oe; and S3.2 producing Mc.imp.1.2. This automatically produces the new version Mc.1.2 of the composite component Mc. The new version of Mc is automatically propagated to produce the new system version Sys.1.4 of the whole system as shown in figure 7.

8. If we assume that another step S4 with a primary input Oa.imp.1.1, with the base version Sys.1.4, then committing this step produces the new version Oa.imp.1.2 and the new version

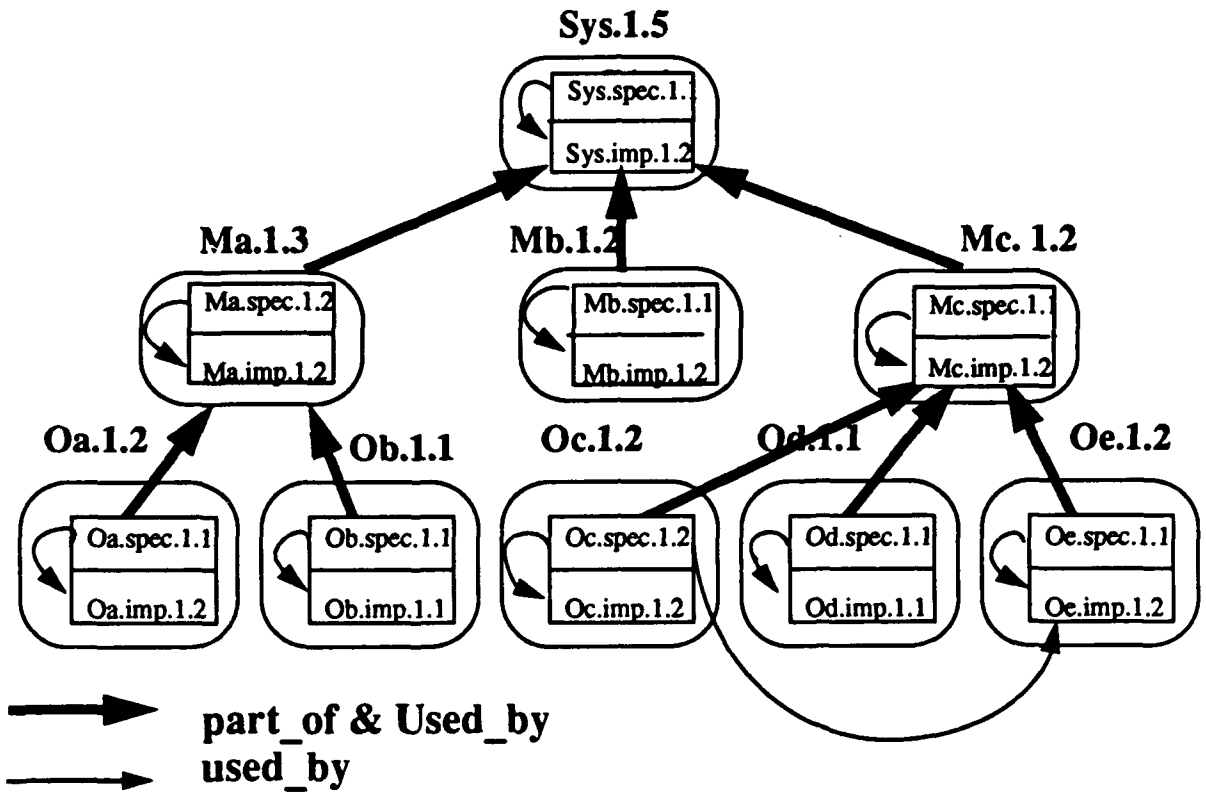


Figure 8 The given system version 5.

Oa.1.2 of the object Oa. This is automatically propagated to produce the new version Ma.1.3 of the composite component Ma and the new version Sys.1.5 of the whole system (as a composite component) as shown in figure 8 below.

9. Now we introduce another step S5 with a primary input Mb.spec.1.1 with the base version Sys.1.4. This step will have three substeps: S5.1 with primary input Mb.spec.1.1 and two induced steps S5.2 with primary input Mb.imp.1.2 and S5.3 with primary input Sys.imp.1.2. Committing this step means that its three substeps are committed, S5.1 produces the new version Mb.spec.1.2, S5.2 produces the new version Mb.imp.1.3, and S5.3 produces the new version Sys.imp.1.3. Now S5.1 and S5.2 lead to producing a new version of the component Mb which is Mb.1.3. To propagate this versioning process to the top component of the system Sys.1.4 we notice that its not the most recent version any more, which leads to producing the new variation of the system Sys.2.5 as shown in figure 9.

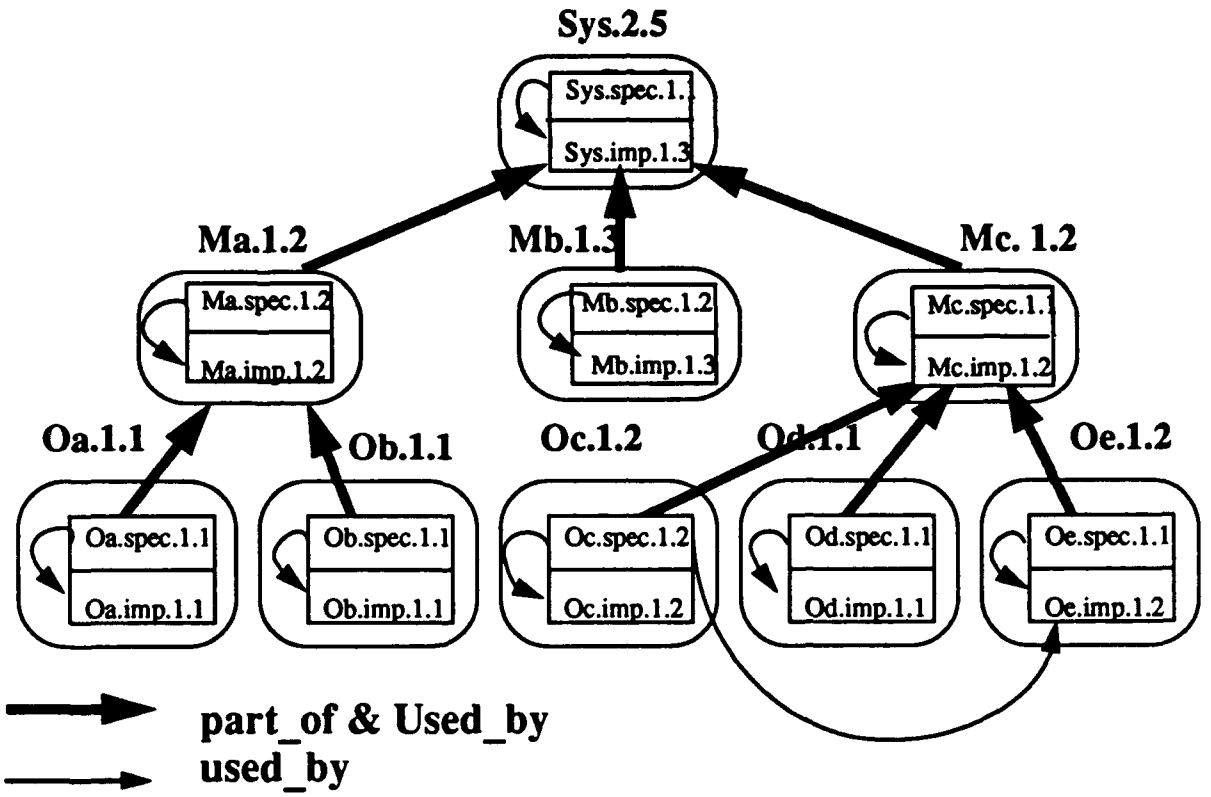


Figure 9 Variation 2 Version 5 of the given system.

6. Conclusion

In this paper we have presented the design database model with its vertical and horizontal graph structures, and the version control and configuration management model and mechanism that simplifies version control and configuration management activities. This mechanism is part of the DMJAS [12] that manages the change process from the moment the primary input of the top level evolution step is entered into the system to committing these changes which produces a new system configuration. The goal of this system is determination of serialization ordering between different steps as well as automation of the assignment of evolution steps to designers. Work on automated support for merging split variation lines is in progress.[18] [19] [20].

LIST OF REFERENCES

[1] Berzins and Luqi, "Software Engineering with Abstractions", Addison-Wesley 1990

[2] Borison E., "A Model of Software Manufacture", in Advanced Programming Environment, Springer-Verlag, 1986, pp. 197-220.

[3] Feldman S. I., "Software Configuration management: Past Uses and Future Challenges" Proceedings of 3rd European Software Engineering Conference, ESEC '91, Milan, Italy, October 1991

[4] Heimbigner D. and Krane S., "A graph Transform Model for Configuration Management Environments", Proceedings of the ACM SIGSOFT/SIGPLAN, Nov. 28-30, 1988.

[5] "IEEE Guide to Software Configuration Management", Std 1042-1987, American National Standards Institute/IEEE, New York, 1988.

[6] Ketabchi M. A., "On the Management of Computer Aided Design Database", Ph. D. Dissertation, University of Minnesota, Nov. 1985.

[7] Luqi, "Software Evolution Through Rapid Prototyping", IEEE Computer Vol. 22, NO. 5. May 1989, pp 13-25.

[8] Luqi, "A Graph Model for Software Evolution", IEEE Transaction on Software Engineering. Vol. 16. NO. 8. Aug. 1990

[9] Mostov I., Luqi, and Hefner K., "A Graph Model for Software Maintenance", Tech. Rep. NPS52-90-014, Computer Science Department, Naval Postgraduate School, Aug. 1989.

[10] Narayanaswamy K. and Scacchi W., "Maintaining Configuration of Evolving Software System", IEEE Trans. on Software Eng. SE-13,3. Mar. 1987, pp. 324-334.

[11] Campbell R. H., Terwilliger R. B. "The SAGA Approach to Automated Project Management", in Advanced Programming Environment, Springer-Verlag, 1986, pp. 142-155.

[12] Badr S. and Berzins V., "A Design Management and Job Assignment System", Technical Report, NPS CS-92-20

[13] Silberschatz A., Stonebraker M., and Ullman J., "Database Systems: Achievements and Opportunities", Communication of the ACM, October 1991/Vol. 34, No. 10, pp. 110-120.

[15] Lie A. et al, "Change Oriented Versioning in a Software Engineering Database", Proceedings of 2nd International Workshop on Software Configuration Management, Princeton, New Jersey, Oct. 24,1989. pp. 56-65.

[16] Gustavsson A., "Maintaining the Evolution of Software Objects in an integrated Environment", Proceedings of 2nd International Workshop on Software Configuration Management, Princeton, New Jersey, Oct. 24,1989. pp. 114-117.

[17] Lobba A., "Automated Configuration Management", Proceedings of IEEE conference on Software Tools 1987. pp. 100-103.

[18] Dampier D., "A Model for Merging Different Versions of a PSDL Program", MS thesis, Computer Science, Naval Postgraduate School, June 1990.

[19] Dampier D., Luqi, "A Model for Merging Software Prototypes", Technical Report, NPS CS-92-014.

[20] Berzins V. "Software Merge: Models and Methods for Combining Changes to Programs", Journal of Systems Integration, vol. 1, no. 2, August 1991, pp. 121-141.

[21] Steigerwald, R. A., "Reusable Software Component Retrieval Using Normalized Algebraic Axioms", PhD dissertation, Naval Postgraduate School, Monterey, CA, December, 1991.

DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Director of Research Administration, Code 08 Naval Postgraduate School Monterey, CA 93943	1
Library, Code 52 Naval Postgraduate School Monterey, CA 93943	2
Egyptian Military Attache 2308 Tracy Place NW Washington, DC 20008	2
Egyptian Armament Authority - Training Department c/o American Embassy (Cairo, Egypt) Office of Military Cooperation Box 29 (TNG) FPO, NY 09527-0051	2
Military Technical College (Egypt) c/o American Embassy (Cairo, Egypt) Office of Military Cooperation Box 29 (TNG) FPO, NY 09527-0051	5
Military Research Center (Egypt) c/o American Embassy (Cairo, Egypt) Office of Military Cooperation Box 29 (TNG) FPO, NY 09527-0051	5
LTC. Salah M. Badr Computer Science Department, Code CS Naval Postgraduate School Monterey, CA 93943	10
Dr. Luqi Computer Science Department, Code CS Naval Postgraduate School Monterey, CA 93943	10

Dr. Valdis Berzins 1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Dr. Mantak Shing 1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Michael L. Nelson, Maj. USAF 1
HQ USCINCPAC / J66
Box 32A
Camp H. M. Smith, HI 96861

Dr.Tarek Abdel-Hamid 1
Administrative Science Department, Code AS
Naval Postgraduate School
Monterey, CA 93943

Dr. J. T. Butler 1
Electrical and Computer Engineering Department, Code EC
Naval Postgraduate School
Monterey, CA 93943